

04IN1023: Grundlagen der funktionalen Programmierung

Nachklausur SoSe 2021

Universität Koblenz-Landau, FB4

Prof. Dr. Ralf Lämmel,

M.Sc., M.Ed. Hakan Aksu

24 September 2021

Name, Vorname	_____
Matrikel-Nr.	_____
Email@uni-koblenz.de
Studiengang	<input type="checkbox"/> BSc Inf <input type="checkbox"/> BSc CV <input type="checkbox"/>
Prüfungsversuch	<input type="checkbox"/> 1 <input type="checkbox"/> 2 <input type="checkbox"/> 3
Prüfungsordnung	<input type="checkbox"/> $\geq PO2019$ <input type="checkbox"/> $< PO2019$

Hiermit bestätige ich, dass ich zur Klausur angemeldet und zugelassen bin.

Eine falsche Angabe wird als Täuschungsversuch gewertet.

Unterschrift: _____

Korrekturabschnitt

Aufgabe	Punkte (0-2)
1	
2	
3	
4	
5	
6	
7	
8	
9	

1 “Simple Algorithmen”

Implementieren Sie die Funktion *doubling*, die einen String einliest. Die Funktion soll jeden zweiten Character verdoppeln.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
    doubling "A" ~?= "A",
    doubling "Hi" ~?= "Hi",
    doubling "Funktion" ~?= "Fuunkktiionn",
    doubling "Programmierung" ~?= "Prroggraammieerungg"
])
```

```
doubling :: String -> String
```

Reference solution

```
doubling [] = []
doubling [x] = [x]
doubling (x:y:xs) = [x] ++ [y] ++ [y] ++ doubling xs
```

2 “Suchen und Sortieren”

Implementieren Sie die Funktion *countFirst*, die einen Integer-Array einliest und die Anzahl der Vorkommisse des ersten Elements zurückgibt.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  firstCount [] ~?= 0,
  firstCount [6] ~?= 1,
  firstCount [5,2,4] ~?= 1,
  firstCount [5,2,5,4] ~?= 2,
  firstCount [4,8,4,1,5,4,75,4,21,5,4,9,4] ~?= 6
])
```

```
firstCount :: [Int] -> Int
```

Reference solution

```
firstCount [] = 0
firstCount (x:xs) = count x xs
  where
    count x [] = 1
    count x (y:ys) = if(x == y) then (1 + count x ys) else (count x ys)
```

3 “Simple Datenmodelle”

Deklarieren Sie einen vereinfachten Datentypen zur Verwaltung einer politischen Partei. Eine *Partei* hat einen Namen, ein Gründungsjahr, den Vorsitzendennamen und eine Liste mit Mitgliedern. Der Name und die Beschreibung sind String-Werte. Das Gründungsjahr ist ein Integer-Wert. Die Mitglieder unterscheiden sich in aktive- und passive-Mitglieder mit jeweils einem Namen. Die restlichen Angaben sind String-Werte. Verwenden Sie bei der Partei und den Mitgliedern geeignete Begriffe mit *type-Synonymen*, um die Lesbarkeit zu verbessern.

Reference solution

```
data Partei = Partei Name GJahr VName [Mitglied]
data Mitglied = Aktiv Name | Passiv Name
type GJahr = Integer
type Name = String
type VName = String
```

4 “Funktionale Datenstrukturen”

Gegeben ist folgender Listen-ADT:

```
data MyList = Empty | Node Int MyList
deriving (Eq, Show, Read)
```

Implementieren Sie die Funktion *add*. Die Funktion *add* erhält zwei Integer-Werte und eine Liste. In die Liste wird an der Index-Stelle (erster Integer) der mitgegebene Wert (zweiter Integer) hinzugefügt. Bei einem negativen Index oder Null (0) wird der Wert vorne hinzugefügt. Bei einem zu großen Index wird der Wert hinten hinzugefügt. Bei den restlichen Index-Werten wird das Element an die entsprechende Stelle hinzugefügt. Die Index-Zählung beginnt bei 0.

Hinweise:

```
> add 0 5 (Node 9 $ Node 8 $ Empty)
Node 5 $ Node 9 $ Node 8 $ Empty
```

```
> add 1 4 (Node 9 $ Node 8 $ Empty)
Node 9 $ Node 4 $ Node 8 $ Empty
```

```
> add 3 7 (Node 3 $ Node 2 $ Node 1 $ Node 9 $ Empty)
Node 3 $ Node 2 $ Node 1 $ Node 7 $ Node 9 $ Empty
```

```
add :: Int -> Int -> MyList -> MyList
```

Reference solution

```
add 0 x l = Node x l
add _ x Empty = Node x Empty
add n x (Node y ys) = if (n <= 0) then (add 0 x (Node y ys)) else Node y (add (n-1) x ys)
```

5 “Funktionen höherer Ordnung”

Implementieren Sie die Funktion *squaresum*, die als Parameter einen Integer-Array erhält. Die Quadrate der einzelnen Werte sollen zusammenaddiert und zurückgegeben werden. Es ist nicht erlaubt einen lokalen Scope zu benutzen (keine Hilfsfunktionen und -variablen). Anonyme Funktionen sind erlaubt!

Hinweise:

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
map :: (a -> b) -> [a] -> [b]
```

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
    squaresum [] ~?= 0
    squaresum [1,2,3] ~?= 14,
    squaresum [6] ~?= 36,
    squaresum [2,3] ~?= 13
])
```

```
squaresum :: [Int] -> Int
```

Reference solution

```
squaresum xs = foldr (\a b -> ((+) b (a*a))) 0 xs
```

Alternative:

```
squaresum xs = foldr (+) 0 (map (\a -> a*a) xs)
```

6 “Typ-Klassen Polymorphismus”

Implementieren Sie zwei Instanzen der Klasse *Animal* für die Datentypen *Cat* und *Dog*.

```
data Cat = Cat Name Age
data Dog = Dog Name
type Name = String
type Age = Float
```

```
class Animal a
  where
    feedingamount :: a -> Float -> Float
```

Implementieren Sie für beide Instanzen eine Funktion, um die Fütterungsmenge zu bestimmen. Die Funktion *feedingamount* erhält ein *Animal* und ein Gewicht als *Float* und berechnet die benötigte Futtermenge als *Float* wie folgt:

- Cat: $0.7 * (\text{Gewicht} - \text{Alter})$
- Dog: $0.2 * \text{Gewicht}$

Hier noch einige Hinweise:

```
sample1 = Cat "Coty" 2.3
sample2 = Dog "Wuff"
```

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  feedingamount sample1 3.8 ~?= 0.9
  feedingamount sample2 8.0 ~?= 1.6 ])
```

Reference solution

```
instance Animal Cat
  where
    feedingamount (Cat _ a) x = 0.6 * (x - a)

instance Animal Dog
  where
    feedingamount (Dog _) x = 0.2 * x
```

7 “Functors & Friends”

Betrachten Sie die Queue-Datenstruktur, die Wertpaare verwaltet.

```
data PairQueue a = Empty | Enqueue (a,a) (PairQueue a)
```

Implementieren Sie die entsprechende Instanz für die Typklasse *Functor* (nur *fmap*).

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  fmap (+1) (Enqueue (1,45) $ Enqueue (2,71) $ Enqueue (3,59) $ Empty)
  ^?= (Enqueue (2,46) $ Enqueue (3,72) $ Enqueue (4,60) $ Empty)])
```

Reference solution

```
instance Functor PairQueue
where
  fmap _ Empty = Empty
  fmap f (Enqueue (i,j) s) = Enqueue ((f i),(f j)) (fmap f s)
```


8 “Unparsing & Parsing”

Implementieren Sie die Funktion `unparseTimetable`, die ein `Timetable` als Parameter nimmt und mithilfe der Bibliothek `HughesPJ` in folgende Form überführt:

- In der ersten Zeile steht das Schlüsselwort 'Meine Wochentermine:' (ohne Hochkomma)
- In jeder weiteren Zeile steht ein Termin in der Form: 'Priority, Tag Von-Bis, Subject;' (Ohne Hochkomma). Der letzte Eintrag enthält kein Semikolon.

```
data Timetable = Timetable [(Priority, Time, Subject)]
type Time = (String, String, String) -- (Tag, Von, Bis)
type Priority = Int
type Subject = String
```

```
sample1 :: Timetable
sample1 = Timetable
  [(1, ("Mo", "12", "14"), "Programming"),
   (1, ("Mo", "14", "16"), "Network"),
   (3, ("Di", "10", "12"), "Database")]
```

`show (unparseTimetable sample1)` liefert folgende Ausgabe:

Meine Wochentermine:

1 Mo 12–14 Programming;

1 Mo 14–16 Network;

3 Di 10–12 Database

Hier sind einige Funktionen aus der `HughesPJ`-Bibliothek:

```
text, int, semi, <>, <+>, $$
```

```
unparseTimetable :: Timetable -> Doc
```

Reference solution

```
unparseTimetable :: Timetable -> Doc
unparseTimetable (Timetable l) = text "Meine Wochentermine:" $$ unparseList l

unparseList :: [(Priority, Time, Subject)] -> Doc
unparseList (x:[]) = unparseElement x
unparseList (x:xs) = unparseElement x <> semi $$ unparseList xs

unparseElement :: (Priority, Time, Subject) -> Doc
unparseElement (x,y,z) = int x <+> unparseTime y <+> text z

unparseTime :: Time -> Doc
unparseTime (x,y,z) = text x <+> text y <> text "-" <> text z
```

9 “Monaden”

Implementieren Sie die Funktion *evalM*, die eine Float-Liste als Parameter nimmt. Die Funktion soll mithilfe des State-Monaden die Summe der Float-Werte bestimmen. Bei jedem positiven Float (inkl. 0) soll die Zustandsvariable um 2 erhöht und bei negativen Float um 1 vermindert werden.

```
runState (evalM [8.3, 1.1, -0.2]) 0  
> (9.2, 3)
```

```
runState (evalM [3, 7, -15, 0, -4, 1, 5]) 0  
> (-3.0, 8)
```

```
evalM :: [Float] -> State Int Float
```

Reference solution

```
evalM [] = do  
  return 0  
  
evalM (x:xs) = do  
  p1 <- evalM xs  
  if (x>=0) then modify (+2) else modify +(-1)  
  return (x + p1)
```